International Journal of Computer Applications (0975 - 8887)
Volume 180 - No.14, January 2018

Prioritizing Dissimilar Test Cases in Regression Testing using Historical Failure Data

Md. Abdur Rahman
Centre for Advanced Research in Science
University of Dhaka
Dhaka, Bangladesh

Md. Abu Hasan
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh

Md. Saeed Siddik
Institute of Information Technology
University of Dhaka
Dhaka, Bangladesh

ABSTRACT

Test case prioritization assigns new order of test cases for detecting regression faults at early. In regression testing when new version is released, all the test cases of both previous and current versions are executed to ensure the desired functionality. This process increases the volume of test cases in regression testing, which is expensive and time consuming. That is why the test cases are needed to be reordered for exploring maximum faults in minimum test cases execution. Usually test case prioritization techniques are designed based on source code coverage, requirements clustering, etc. Most of these techniques contain the similarity relationship among the test cases. However, similarity based technique may stuck in local minima. To overcome the limitation of similarity based prioritization, this paper proposed the dissimilar clustering based approach using historical data analysis to detect maximum faults. In this approach, dissimilar test cases placed in the top of the test suites and executed earlier than similar test cases. Proposed scheme is evaluated using well established Defects4j dataset, and it has reported that proposed strategy performs 54.95%, 41.83% and 7.00% better than untreated (normal ordering), random and similarity cluster based prioritization methods respectively.

Keywords

Prioritization, Dissimilarity, Clustering, Historical Data

1. INTRODUCTION

Test case prioritization reschedules test cases for execution to increase the regression testing effectiveness. The regression testing revalidates the software to ensure that the newly introduced code does not affect existing software adversely [1]. This revalidating procedure runs both previous and newly introduced test cases, which is costly in terms of expense and time [2]. It has been experimented that more than 50 days are required to test 20,000 lines of code [2]. Therefore, diverse methods are introduced to improve regression testing performance based on cost effectiveness, which are categorized into Test Suite Reduction (TSR), Test Case Selection (TCS), and Test Case Prioritization [3], [2]. TSR removes redundant code coverage test cases, where TCS picks only those test cases that covers changed portion of the software [2]. While TSR and TCS reduce regression testing reliability by omitting test cases which can also detect faults, test case prioritization reorders test cases to meet testing goals, such as rate of fault detection, source code coverage, and quick feedback [4].

Several test case prioritization methods has been proposed to increase fault detection rate in regression testing which may categorized into requirement coverage [3][5], code coverage [6][1], probabilistic based [4][7], history based [8][9], etc. Using requirement similarity clustering Arafeen et al. proposed prioritization method to investigate regression testing efficiency [3]. Different software artifacts such as requirements, design diagrams and source code are used for prioritization [5]. Modified genetic algorithm has also been used for prioritizing test cases to improve code coverage, where control flow graph was generated from the selected program [6].

History based prioritization approach was proposed to evaluate the performance under severe time and resource constraints [10]. Customer and developer assigned priority are also used to provide test cases priority [8]. Similarity based prioritization using historical failure data was proposed to rank new test cases in regression testing [11]. Bayesian Network and its modified test case clustering version has been used for prioritization to improve fault detection rate [4][7]. However, test case clustering on code coverage profile contains risk to detect similar fault consecutively. As a result, get similar priority that leads to decrease the efficacy of fault detection rate. This scenario is illustrated by following example, where nine test cases cover nine faults as shown below.

$TC1 :< f1, f4, f5, f6, f7 >, TC2 :< f2, f3, f4, f5, f6 >, TC3 :< f2, f3, f4, f7, f8 >, TC4 :< f2, f3, f7, f8, f9 >, TC5 :< f1, f6, f7, f9 >, TC6 :< f1, f5, f6, f7.f9 >, TC7 :< f1, f2, f3, f4, f8 >, TC8 :< f1, f6 >, TC9 :< f1, f2, f3, f4, f6 >$.

$Cluster - 1 :< TC1, TC5, TC6, TC8 >$,
$Cluster - 2 :< TC2, TC3, TC4 >$,
$Cluster - 3 :< TC7, TC9 >$.

The majority test cases in cluster-1 detect faults f1, f6 and f7, but f2, f3 and f8 faults are uncovered. On the other hand, test cases in cluster-2 cover faults f2 and f3, excluding fault f1. Both test cases in cluster-3 detect faults f1, f3, f4 and f9, but faults f5 and f7 are out of coverage. In this example, same faults covered by more than one clusters. However, any single cluster cannot cover all faults, which is the limitation of similar cluster based test case prioritization which shares same properties.

This paper presents dissimilar test case clustering technique to overcome the similar clustering based prioritization problem. In the proposed approach, present version test cases are used to generate call dependency graph which forms similar test case cluster.

On the other hand, both present and previous version test cases are considered to generate failure history call dependency graph. Intra cluster test cases are prioritized on test case degree of connectivity and failure history. At the end, dissimilar clusters are formed selecting top test cases from similar clusters to cover maximum varieties of faults. The preliminary investigation about the dissimilar cluster based test case prioritization has been presented in [9]. In that paper, a short methodology description has been addressed for similar and dissimilar clustering. In Addition, no direction has been presented for orphan test suite handling and dissimilar test suite formation. This research introduces the complete dissimilar cluster based prioritization using historical failure data including the solution of several limitations in our previous work.
The proposed technique has been experimented with Defects4j dataset [12], and experimental results are evaluated with different significant strategies named as random prioritization, similarity based prioritization and untreated prioritization. The investigated result shows that dissimilarity based prioritization using failure history performs 54.95%, 41.83% and 7.00% better than untreated, random and similarity based prioritization methods respectively.
The rest of this paper is organized as follows, Section 2 describes the literature review. Section 3 and 4 illustrated the proposed method and result analysis respectively. Finally section 5 concludes this paper with future research direction.

2. LITERATURE REVIEW

Test case prioritization technique rearranges test case ordering to maximize testing objectives, such as improving fault detection rate, reducing execution time, etc. Because of significance in practice, several prioritization techniques have been developed, which may categorized in requirement coverage [3][5], source code coverage [6][1], probabilistic based [4][7], and history based [8][9]. Several prioritization techniques are discussed in this section.

2.1 Requirement Coverage based Prioritization

Requirement similarity clustering has been used for investigating the efficiency of regression testing in test case prioritization [3]. Term document matrix has been generated from software requirements, which lead to k-means clustering algorithm. Clusters of test cases was formed using requirements test case mapping traceability matrix. Execution sequence of clusters was ordered using code modification information and clients requirement priority. Result indicates that the strategy improved effectiveness of prioritization.
Test case prioritization using the collaboration of different software artifacts such as requirements, design diagrams and source code has been proposed by [5]. This scheme overcomes the limitation of traditional single SDLC phase consideration in software testing. In their approach, requirements connectivity, design inter-dependency and code metrics are collected, and multiplied by their weight for measuring final priority of test case. The experimental analysis figure out that use of collaborative information in test case prioritization was significant. However, direction was undeclared for assigning weight to SDLC phases, and the result would be more effective by incorporating historical failure data.
Dusica et al. proposed a multi-perspective prioritization framework in time-constrained environments for faster fault detection [13]. This scheme considers execution time, inter dependent functionality, failure impact and frequency factors for selecting the multi-perspective values. This approach prioritizes test cases having maximum inter dependence functional coverage, failure impact and frequency. Though, this technique used multiple factors for prioritizing test cases, dissimilar based test suite selection has not been considered to detect different types of fault at early execution.

2.2 Code Coverage based Prioritization

Rothermel et al. demonstrated a number of prioritization strategies to improve fault detection rate in regression testing [2]. They performed empirical studies to evaluate quality, importance and quantity of the rate of fault detection of different techniques. In their analysis seven different programs with both current and modified versions were used. Empirical result indicated that the proposed approach detect fault efficiently at early. Results also showed that code coverage based prioritization performed better than additional branch coverage, where total statements coverage approach performed efficiently than additional statement coverage strategy.
Patipat et al. implemented a modified genetic algorithm for prioritizing test cases to improve code coverage [6] in regression testing. A control flow graph was generated from the selected program, which derived to get decision graph. Test cases were generated randomly from the decision graph according to the population size in genetic algorithm. Test suites was formed using selected test cases measuring conditions covered by each test case. The fitness value of every test case was determined based on the coverage information, which is used to rank the suites. Finally test suites were ranked using fitness value. The result showed that modified genetic algorithm performed better than Bee Colony and random approaches. However, generating complex decision graph for large scale software may overhead of this approach.
Rothermel et al. presented a prioritization approach using control flow graph of procedure or program and its modified version [1]. Using these graphs the method selects the test cases having changes from the original test cases. The proposed approach chooses test cases that may execute for the first time in new version. The approach can manage test case selection of regression testing both for single and groups of interacting methods. The main contribution is reduction of execution time required in regression testing. Another contribution is saving of time and costing. The limitations of proposed method are time estimation, artifacts construction that can lead to a scope of work for further research on this field.

2.3 Probabilistic based Prioritization

Bayesian Network based prioritization framework has been proposed to improve fault detection rate by Siavash et al. [4]. The proposed methodology used the most known statistical probability principle with the utilization of bayesian network. The framework took program modification, tendency of fault occurrence and test case coverage information as a input and produces the probability of test case as output. In first step, source code was taken into action in order to extract several proofs from this source code. After extracting the evidence, the second step took all necessary information to an individual bayesian network model to accomplish. In the final step, statistical probabilistic theorem has been applied to measure the probability success. The result showed that proposed method performed better than other implemented techniques such as normal ordering and random ordering, if the available faults are remarkable. However, in this proposed approach, several test cases may indicate similar faults in execution.
Xiaobin et al. proposed an enhanced Bayesian Network (BN) based method for prioritization, where test cases are clustered using method level coverage matrix [7]. Intra cluster test cases are prioritized based on their fault detection probability by BN approach. Source code change information and class level coverage matrix are fed as input to get failure probability as output. The result shows

that the improved BN scheme is more effective than normal BN model for test case prioritization. However, failure history which is effective to detect fault in regression testing has not been considered for test case prioritization in the proposed strategy.

2.4 History based Prioritization

History based regression test case prioritization using requirement priority was proposed where, customer and developer assigned priority are used to provide test cases priority [8]. Initially prioritized test cases were executed and fault detection history was recorded for next execution. The differences of requirement priority between two adjacent test cases are used to reorder the execution sequence dynamically. An industrial experimentation has been performed in order to evaluate the technique, and result analysis has shown that the history based prioritization method improved testing effectiveness and fault detection ability than random and other common methods. However, the efficiency of this technique depends on the assigned requirement priority by customers and developers which may biased and affect the deserved prioritization effectiveness.
Jung et al. proposed a history-based test prioritization approach to evaluated the performance under severe time and resource constraints of several Regression Test Selection (RTS) techniques [10]. If the resource constraints are not considered, prioritization performance may become unpredictable. The cost and benefits of several RTS techniques under two different software evolution models has been investigated. Results strongly supported that regression testing might have to perform differently in constrained environments than non-constrained ones. In addition, historical information might be useful in reducing costs and increasing the effectiveness of long running regression testing process.

2.5 Fault based Prioritization

Similarity based prioritization approach using historical failure data was proposed to rank new test cases in regression testing [11]. Sequence of method call by previously failed and new test cases has been generated to measure distance value from each other, which is used to form similarity matrix in descending order. Similarity between test cases defined by calculating their method calling sequence. The experimental result stated that the proposed similarity based approach performed more efficiently as it can effectively prioritize test cases compared to other traditional strategies. However, there is no direction when faults history is absent or not adequate.
Do Hyunsook et al. introduced an experimental measurement of test case prioritization using fault mutation strategy [14]. This paper represents prioritization effectiveness for hand seeded faults and mutant faults. The fault mutation was performed by changing operators such as logical operator, arithmetic operator, overriding variable changing etc. The experimental analysis showed that prioritization performance depends on some independent and dependent variables such as test suite granularity, fault detection rate, etc.
The analysis of existing prioritization approaches shows that different prioritization strategies have been proposed for regression testing such as code coverage, clustering, historical data etc. Incorporating historical data with similarity clusters is recently introduced where similar test cases are stuck on uniform faults and inefficient in detection of variety faults. However, no direction has been found yet to detect dissimilar faults using historical data, which may increase variant fault detection rate in regression testing.

3. PROPOSED METHODOLOGY

The dissimilar based test case prioritization framework is proposed to improve the performance of regression testing by executing dissimilar test cases earlier. This method selects and executes more significant test cases earlier with intent to cover divergent faults. In this approach, similar and dissimilar test cases are clustered using function call connectivity, which are depicted at call dependency graph. Similar cluster test cases are internally ordered based on failure history and connectivity among other test cases. Finally top test cases are selected from every cluster to form a dissimilar test suite. This process is iterated until all the test cases are covered, which makes the desired prioritization sequence of this approach. The proposed method is divided into following five steps.

Step 1: Call Dependency Graph (CDG) Generation
Step 2: Similar Test Case Cluster Formation
Step 3: Failure History CDG Generation
Step 4: Intra Cluster Elements Prioritization
Step 5: Dissimilar Test Cases Cluster Formation

Detail descriptions of the above steps are given in the following sub-sections.

3.1 Step 1: Call Dependency Graph (CDG) Generation

Function call dependency graph represents the list of interconnected test cases in terms of function call, which is used to find similarity among test cases. Similarity between two test cases defines their degree of relationship, which is measured with the CDG. The value of presence or absence of connectivity between two test cases is presented in CDG. The connectivity value is Boolean, which is either connected (1) or disconnected (0).

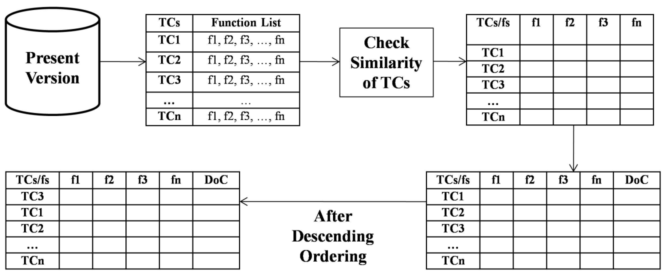

Fig. 1. CDG of present version test cases

Present version test cases are taken as input to find out the list of connected test cases in term of function call. After that, function call dependency graph *G (V, E)* is generated, where test cases and function calls between two test cases are presented as vertex *(V)* and edges *(E)* respectively. Degree of connectivity for each test case is computed on the basis of CDG connections. The CDG generation process is illustrated at Figure 1.

3.2 Step 2: Similar Test Case Cluster Formation

Similar clusters are formed based on the similar test cases by considering fault coverage. Among various clustering algorithm [15], K-means [16] is used for similar cluster formation, where the preliminary number of cluster is defined by the square root ceiling value of total test cases *(n)*. Test cases are decreasingly ordered based on their degree of connectivity using CDG, and assigned the top n test cases as the cluster heads. Initially clusters are formed using only the header test case.

pseudocode [h] **Input:** Call Dependency Graph $G(V, E)$
Output: Test Case Cluster (C) Similar Cluster Formation [1] **Begin** $T_s \leftarrow$ Sorted Test Suite on degree of connectivity Number of cluster $(n) = \left\lceil \sqrt{Ts} \right\rceil + 1$ $C \leftarrow \{\}$
all $i \in \{1...(n-1)\}$ $C_i \leftarrow C_i \cup \{t_i\}$ $t_i \in T_s$ $L_i \leftarrow \phi$ $C_j \in C$ t_j $is\ connected\ to\ anyC_i$ $L_j \leftarrow L_j \cup \{C_i\}$ $C \leftarrow C - \{C_i\}$ $L_j = \phi$ $C_{iso} \in C_{iso} \cup \{t_i\}$ **else if** $|L_j| = 1$ **then** $C_i \leftarrow L_j$ $C_i \leftarrow max(L_j)$ **End**

The similar cluster formation process in presented in Algorithm 1, which takes call dependency graph G(V,E) as input and generated similar cluster (C) as output. In the algorithm, n number of empty clusters (C) is created (line 4) and these clusters are initialized with n number of top test cases (line 5, 6, and 7). An empty mapping (L_j) for j^{th} test case is created (line 9). Test cases connectivity with cluster head are checked (line 10-15) to form similar cluster. Disconnect, single connection and multiple connection are presented in line 19, 20 and 21 accordingly.

After assigning the headers, remaining test cases are required to assign inside these clusters based on their degree of connectivity. If any test case connects to single cluster head, it is assigned to that cluster. On the other hand, for multiple connectivity; it is assigned to the cluster whose degree of connectivity is maximum. Test cases having no connectivity are inserted inside an orphan cluster, which is previously created containing disconnected test cases. This process will be continued until all the test cases are assigned. In the next step, orphan cluster test cases are assigned to other clusters by measuring the connectivity with members of other clusters.

This approach is continued until orphan cluster members get their cluster identity. The remaining orphan cluster test cases those do not get access inside any clusters are considered as isolated cluster member. Thus, a number of cluster is formed those are two categories defined as similar connectivity clusters and isolated cluster.

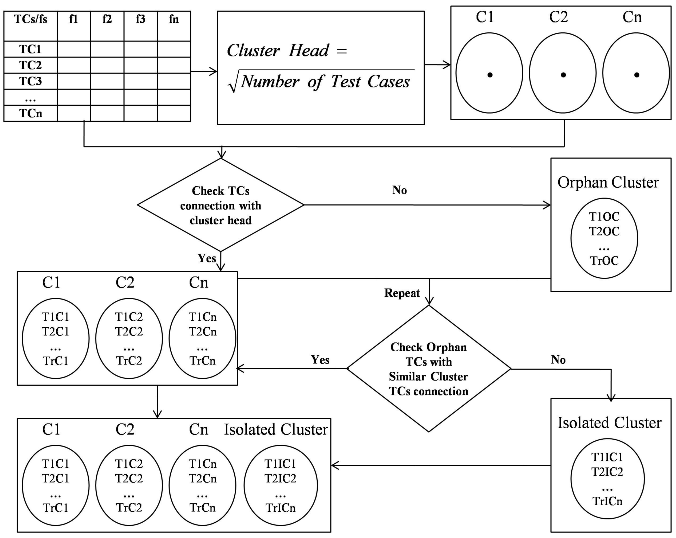

Fig. 2. Overview of Similar Cluster Formation

For example of 9 test cases with the degree of connectivity are $< TC1, 2 >, < TC2, 3 >, < TC3, 1 >, < TC4, 3 >, < TC5, 4 >, < TC6, 1 >, < TC7, 2 >, < TC8, 1 >, < TC9, 2 >$. There should have 4 clusters including 1 isolated cluster. The test cases are sorted in descending order according to their degree of connectivity and the top three test cases $< TC5, TC2, TC4 >$ are selected as cluster head. Here, TC1 is connected with all head, however, maximum connection exists with TC5, as a result, TC1 is assigned insider Cluster-1. Where TC7 is connected with TC2 and TC4, also degree of connectivity is equal; as a result, it is assigned randomly. However, TC3 is not connected to any header, so it is assigned in isolated cluster. The whole overview of similar cluster formation is shown in Figure 2, and the example test cases will form the following clusters.

Cluster-1:$< TC5, TC1, TC9, TC8 >$
Cluster-2:$< TC2, TC7, TC6 >$
Cluster-3:$< TC4 >$
Isolated Cluster:$< TC3 >$

3.3 Step 3: Failure History CDG Generation

Failure history call dependency graph is generated using both present and previous version test cases. Because, previously failed test cases have highest possibility to fail in subsequent execution in regression testing [11]. A list of similar test cases is forwarded from that failure history call dependency graph. The listed similar tests cases with their corresponding number of fault detection history are profiled as previous execution history. This profile will be used for assigning priority to present version test cases in Step 4. Similarity between previous and present version test cases is measured using inter function connectivity (degree of connectivity) of both versions which are presented in Figure 3. Basic CDG generation process is described elaborately in Step 1.

3.4 Step 4: Intra Cluster Elements Prioritization

Intra cluster test cases are prioritized using failure history and degree of connectivity of previous and current version test cases respectively. The priority of every test case is calculated using their corresponding degree of connectivity and fault detection ability, which is depicted in Equation (1). To calculate individual test case's priority, total degree of connectivity is divided by total number of test cases, and then the outcome is added with number of detected faults. To normalize the result, degree of connectivity is divided by the total number of test cases. Random ordering will be applied, when similar priority is calculated for multiple test cases.

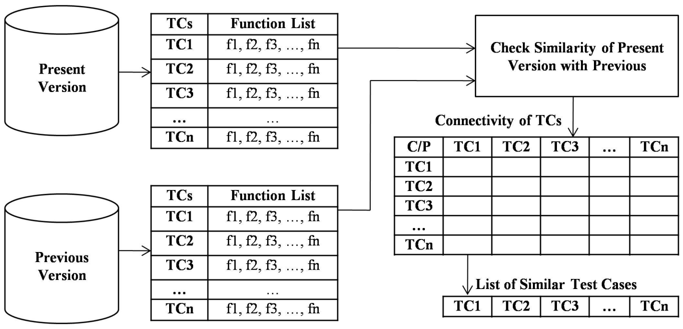

Fig. 3. Failure History CDG Generation

pseudocode [h] Internal Test Case Ordering **Input:** Test Case Cluster Ci, failure history L_f

Output: Prioritize test case P [1] **Begin** $T \leftarrow$ $read\ all\ test\ cases\ from\ every\ clusterC_i$ $DoC \leftarrow$ $calculate\ degree\ of\ connectivity\ ofT$ $L_f \leftarrow$ $number\ of\ faults\ for\ each failure\ test\ cases$ $t_i \in T$ $t_i \cup L_f$ $P_i \leftarrow L_f\,[ti] + \frac{DoC[ti]}{NumberofTc}$ $P_i \leftarrow \frac{DoC[ti]}{NumberofTc}$
End

The internal test case ordering of similar clusters *(C_i)* is represented in Algorithm 2, where *(C_i)* and profiled failure history *(L_f)* takes as input and prioritize test cases *(P)* are generated as output. Priority of i^{th} test case *(P_i)* is measured by number of detected faults *($L_f[t_i]$)* and normalized degree of connectivity ($DoC[t_i])/No.\ of\ Tc$) (Algorithm 2 line 6-7). Where, test cases dissimilar to previously fail are prioritized by normalized degree of connectivity only (Algorithm 2 line 9).

$$TC_{priority} = \frac{Degree\quad of\quad Connectivity}{Number\quad of\quad Test\quad Cases} + NumberofFaults \tag{1}$$

For given example of 9 test cases in Step 2, Degree of Connectivity (DoC) and failure history are presented as follows:

Degree of Connectivity: $< Test\quad Case, DoC >=< TC1, 2 >$ $,< TC2, 3 >, < TC3, 1 >, < TC4, 3 >, < TC5, 4 >, <$ $TC6, 1 >, < TC7, 2 >, < TC8, 1 >, < TC9, 2 >$

Failure History: $< Test\quad case, Number\quad of\quad Faults >=<$ $TC2, 1 >, < TC4, 1 >, < TC5, 2 >$

Several clusters are formed using these test cases and internally prioritized using Equation (1) which are as follows.
Cluster-1:$< TC5(2.44), TC1(0.22), TC9(0.22), TC8(0.11) >$
Cluster-2:$< TC2(1.33), TC7(0.22), TC6(0.11) >$
Cluster-3:$< TC4(1.33) >$
Isolated Cluster:$< TC3(0.11) >$.
Test cases are formed similar cluster and ordered in descending based on their calculated priority, where dissimilar cluster formation will be described in the subsequent step.

3.5 Step 5: Dissimilar Test Cases Cluster Formation

Dissimilar test case clusters are formed using internally prioritized similar clusters from Step 4. Test cases having equal level of priority form a distinct dissimilar cluster together. Where, level of priority denotes the internal position of a test case inside a cluster. For example, first dissimilar cluster will be formed with test cases having first level of priority of prioritized similar clusters. This dissimilar cluster generation process will be continued until two similar clusters contain at least one test case. If only one similar cluster remains for processing, the dissimilar cluster formation will be terminated. The remaining similar cluster forms a separate cluster. The whole process of dissimilar cluster formation is depicted in Figure 4, and example of it is given below.
For example, intra prioritized test cases in Step 4 will form the following dissimilar clusters. The dissimilar clusters test cases are internally ordered using their corresponding priority which is the subsequent of the example of Step 4.

Dissimilar Clusters
$Cluster{-}1 :< TC5(2.44), TC4(1.33), TC2(1.33), TC3(0.11) >$ $, Cluster - 2 :< TC9(0.22), TC6(0.11) >, Cluster - 3 :<$ $TC7(0.22), TC1(0.22) >, Cluster4 :< TC8(0.11) >$

pseudocode [h] Dissimilar Cluster Formation **Input:** Similar cluster C_s
Output: Dissimilar clusters C_{dis} [1] **Begin** $C_s \leftarrow$ Read all similar clusters $C_n \leftarrow$ Number of similar clusters $C_n \leftarrow \{\}$
$i\ to$ 1... $|C_n|$ all clusters $C_i \leftarrow C_s$ $C_{dis} \leftarrow C_{dis} \cup C_i\{t_i\}$ $C_i \leftarrow$ $C_i - \{t_i\}$ **End**

The dissimilar cluster formation process is presented in Algorithm 3, which takes similar clusters (C_s) as input and generated dissimilar clusters (C_{dis}) as output. The similar clusters are ordered based on number of assigned test cases and the second highest cluster element number is used to form empty dissimilar clusters (C_{dis}) (Algorithm 3, line 4). Because, at least two non-empty similar clusters are required processing dissimilar cluster formation. Top test cases from each similar cluster are selected to create dissimilar clusters (Algorithm 3 line 5-7). This process will iterated until at least two similar clusters are remains with element.

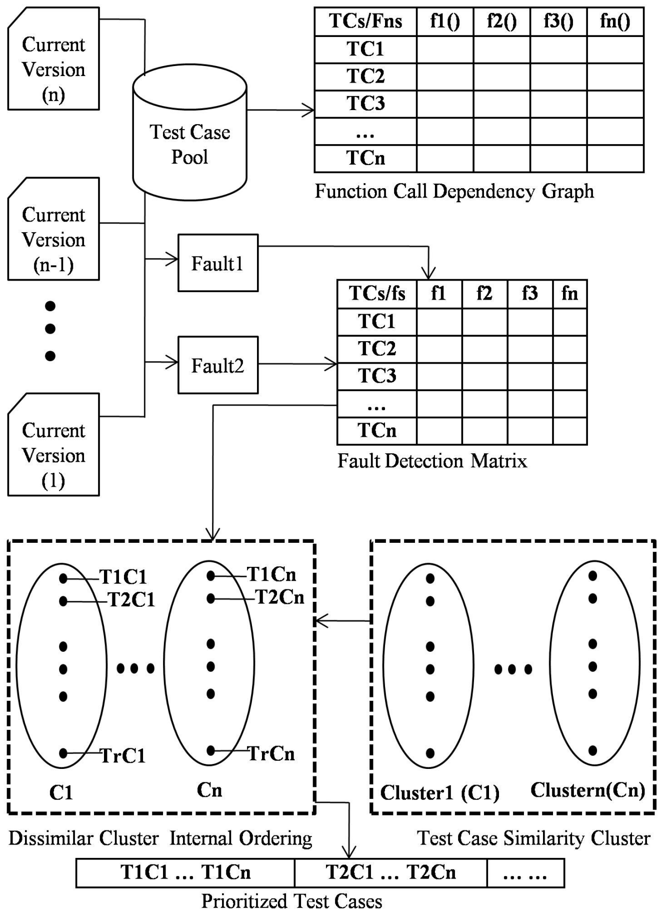

Fig. 4. Dissimilar Cluster Formation Overview

The proposed dissimilar clustering prioritization approach is implemented to re-order test cases in such a way, faults can be revealed in early of the testing phase. This dissimilar clustering strategy uses function call dependency connectivity graph G (V, E) to identify dissimilarity between the test cases. The test cases are ranked in descending order on the failure history and degree of connectivity value. The top ordered test cases indicate early fault detection, and should be executed earlier than other test cases.

4. EVALUATION

The detailed experimental study, dataset and evaluated results are demonstrated in this section. Also, the effectiveness of proposed dissimilarity based test case prioritization strategy is evaluated.

4.1 Experimental Prioritization Method

Proposed dissimilarity based test case prioritization method is compared with four different prioritization techniques named as Untreated Test case Prioritization (UTP), Random Test case Prioritization (RTP) and Similar Test case Prioritization (STP). Where, UTP refers to normal ordering of test cases of a test suite and RTP denotes random order of test cases of a test suite. On the other hand, STP refers to similarity based test case clustering and ordered with fault detection average weight.

4.2 Research Questions

In this thesis following research questions has been investigated:

RQ 1: Can the proposed method improve prioritization effectiveness compare to UTP?

RQ 2: Can DTP technique performs better than RTP?

RQ 3: Can DTP strategy detect faults at early of testing phase compare to STP?

4.3 Dataset

Two versions of three different projects named as JodaTime, Closure, and Chart from well reputed Defects4j dataset are used for experimental analysis in this paper [12]. Defects4j dataset contains 20,109 tests and 357 bugs in each individual projects. Each version of project contains buggy and fixed code segments with corresponding test cases. All the test cases are written in Junit test method. Projects which are used as dataset for this paper experimentation from Defects4j are shown in below.

Table 1. Used Dataset

Identifier	**Project name**	**Number of bugs**	**Number of Test Cases**
Closure	Closure compiler	133	7,927
Chart	Jfreechart	26	2,205
JodaTime	Joda-Time	27	2,245

4.4 Environment Setup

The research work evaluation has been performed on a single personal computer having 2.5 GHz core i5 CPU and4GB memory running the Ubuntu14.04 LTS version operating system. To run Defects4j java 1.7, perl 5.0.10, git 2.10.1, and SVN 1.9.5 have been installed. LAMPP server has been installed in order to execute php scripts, which are used to generate dissimilar test suites.

4.5 Evaluation Metric

In test case prioritization technique, standard measurement metric named as APFD (Average Percentage of Faults Detection) is used to calculate the average fault detection percentage for the test suite [2]. The limit of APFD result is 0 to 100, where higher number indicates faster fault detection rate. Let a test suite T containing n test cases; F denotes a set having m faults which is revealed by test suite T. TFi is the position number of earliest test case of test suite T which detects fault i. The APFD is calculated using the following Equation (2).

$$APFD = 1 - \frac{TF_1 + TF_2 + TF_3 + ... + TF_m}{n * m} + \frac{1}{2n} \quad (2)$$

4.6 Result Analysis

The defects4j dataset mentions which test case actually fails in the current version [12]. So, the rank of the first failing test cases provided by different approaches in each version is compared. The mutation is done to increase the faults as there is only one fault per version in defects4j. In the rest of this section research questions answer is discussed.

Table 2. APFD based on Various Percentage of Test Execution

Datasets	Methods	**APFD**				
		Input				
		20%	**40%**	**60%**	**80%**	**100%**
Closure v2	UTP	36.53	36.53	42.30	100	100
	RTP	19.61	45.38	57.69	78.07	100
	STP	98.07	98.07	98.07	100	100
	DTP	100	100	100	100	100
Closure v3	UTP	34.61	40.38	46.15	100	100
	RTP	28.84	56.92	68.07	97.69	100
	STP	86.53	96.15	100	100	100
	DTP	94	100	100	100	100
Chart v2	UTP	10.52	10.52	42.10	100	100
	RTP	13.68	32.63	64.21	88.37	100
	STP	26.31	26.31	57.89	100	100
	DTP	100	100	100	100	100
Chart v3	UTP	0	0	29.16	83	100
	RTP	20	36.65	62.50	73.32	100
	STP	54.16	87.50	100	100	100
	DTP	79	79	79	96	100
Joda Time v2	UTP	0	20	20	100	100
	RTP	10	18	64	94	100
	STP	80	90	100	100	100
	DTP	80	100	100	100	100
Joda Time v3	UTP	0	28	28	100	100
	RTP	17	45.60	63	91	100
	STP	72	88	92	100	100
	DTP	80	90	100	100	100

4.6.1 Experimental results for RQ1.

To evaluate RQ1, the result of UTP and DTP is compared. In the experiment, for every project of two versions, DTP performs better than UTP which is shown in the Table 2. For example in Chart v3 dataset, APFD values of DTP for different input size are 79%,79%,79% and 96% where UTP APFD values are 0%, 0%, 29.16% and 83% respectively.
The various APFD results for UTP and DTP are averaged and figured out at Figure 5, where the area under the curve represents the APFD, and it shows that APFD of proposed DTP method is 88.54%, which is higher than UTP 40.77%.

4.6.2 Experimental result for RQ2.

RTP and DTP prioritization methods are compared to answer RQ2. According to the Table 2, APFD of RTP is lower than proposed method DTP. In the experimental analysis for dataset Jodatime v3 the DTP APFD values for different input sizes are 80%, 90%, 100%, 100% and 100% which

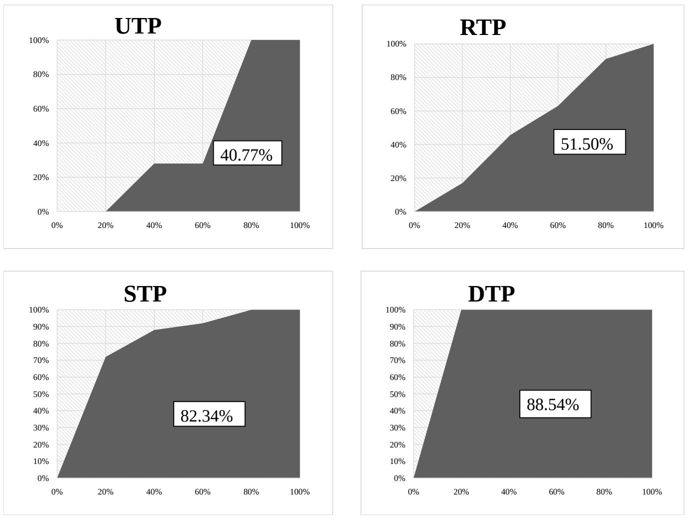

Fig. 5. Average APFD of various prioritization schemes

Table 3. APFD Comparison

Dataset	Version	APFD (%)			
		UTP	RTP	STP	DTP
Closure	v2	36.91	49.00	52.98	87.74
	v3	28.24	45.90	84.43	81.19
Chart	v2	46.73	54.46	95.55	95.10
	v3	47.91	59.77	87.23	90.32
Joda Time	v2	41.32	49.61	90.84	92.12
	v3	43.52	50.26	83.02	84.79
	Average	40.77	51.50	82.34	88.54

are always higher than RTP APFD values 17%,45.60%,63%, 91% and 100% respectively. The various APFD results for RTP and DTP are averaged and figured out at Figure 5. According to Figure 5, the area under the curve represents the APFD, and it shows that our DTP method APFD is 88.54%, which is higher than RTP APFD 51.50%.

4.6.3 Experimental results for RQ3. Test case prioritization effectiveness of STP and DTP is evaluated to get the RQ3 answer. In this paper the proposed method DTP detects faults earlier than STP in 66.67% cases which is measure according to the APFD metric. According to the Table 2 for dataset Jodatime v3, the calculative APFD for proposed DTP method for various input size are 80%,90%,100%,100%,100% where STP APFD is 72%,88%,92%,100% & 100% respectively.
The various APFD results for STP and DTP are averaged and figured out at Figure 5, where the area under the curve represents the APFD, and it shows that our DTP method APFD is 88.54%, which is higher than STP APFD 82.34%.

4.7 Discussion

Table 3 and Average APFD figure 5 show the results of experiment on three different projects. The table shows that the performance ranking of four different test case prioritization techniques are $UTP < RTP < STP < DTP$ for three projects datasets based on APFD metric calculation. APFD values of UTP and RTP techniques are always lower than proposed DTP technique; and 66.67% cases DTP have higher APFD value than STP method. The box-plot of figure 6 represents the average APFD value of each test

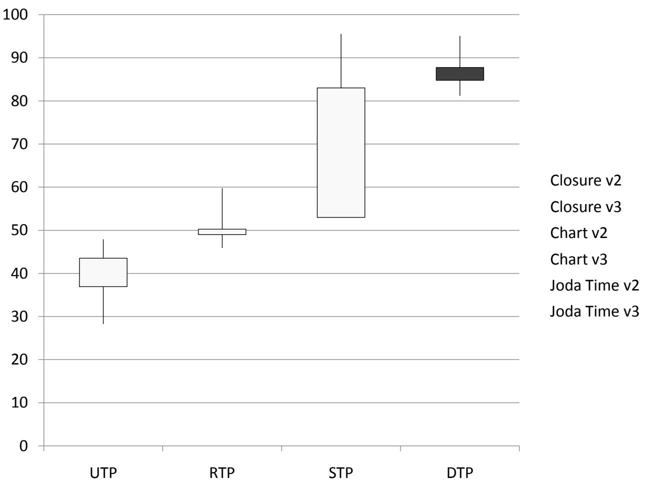

Fig. 6. Box plot of Several Prioritization Techniques

case prioritization method of three projects where DTP has higher value 88.54% compare to UTP, RTP, and STP 40.77%, 51.50%, and 82.344% value respectively. That means the dissimilar test suite selection can reduce the test case execution time and maximize the fault detection rate.

4.8 Threats to Validity

Without the prior concern of internal dependent variables and external conditions threats may raise in prioritization system. The potential threats to the validity of the studies will be related to the regression test case dataset. More specifically, the absence of previous version test case and failure history may affect the final accuracy, which is one of the threat of proposed dissimilarity based approach.

5. CONCLUSION

This paper presents a dissimilarity based test case prioritization using historical failure data. The main contribution of this work is to reorder test cases for improving performance of prioritization using dissimilar test cases and previous testing failure information. The proposed technique combines failure history, and regression versions' test cases (current and previous) for prioritization. At first similarity is measured between two test cases using call dependency graph which leads to form several similar clusters. Intra cluster test cases are ordered using previous testing failure information. Finally test cases are selected from every distinct cluster to create the new dissimilar test suite. This technique has been experimented on Defects4j dataset which includes three java projects named as JodaTime, Closure, and Chart. The proposed prioritization framework is evaluated with the well known measurement metric named as Average Percentage of Fault Detection (APFD). Performance of proposed dissimilar approach has been compared with untreated, random and similar clustering scheme, and found that 54.95%, 41.83% and 7.00% better result respectively. The investigated results reveal that the use of dissimilar approaches proves the effectiveness of prioritization in terms of higher rate of fault detection, compared with existing untreated, random and similar approaches. Dissimilarity based clustering still has a certain extent of similarity among the intra-cluster test cases. Minimizing those similarity will be the future direction of this research.

6. ACKNOWLEDGEMENT

This research has been partially supported by Information and Communication Technology Division (ICTD), Peoples Republic of Bangladesh Research Grant No-56.00.0000.028.33.069.16-893.

7. REFERENCES

[1] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, pp. 173–210, 1997.

[2] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[3] M. J. Arafeen and H. Do, "Test case prioritization using requirements-based clustering," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pp. 312–321, IEEE, 2013.

[4] S. Mirarab and L. Tahvildari, "A prioritization approach for software test cases based on bayesian networks," *Fundamental Approaches to Software Engineering*, pp. 276–290, 2007.

[5] M. S. Siddik and K. Sakib, "Rdcc: An effective test case prioritization framework using software requirements, design and source code collaboration," in *17th International Conference on Computer and Information Technology (ICCIT)*, pp. 75–80, IEEE, 2014.

[6] P. Konsaard and L. Ramingwong, "Total coverage based regression test case prioritization using genetic algorithm," in *12th Int. Conf. on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, pp. 1–6, IEEE, 2015.

[7] X. Zhao, Z. Wang, X. Fan, and Z. Wang, "A clustering-bayesian network based approach for test case prioritization," in *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, vol. 3, pp. 542–547, IEEE, 2015.

[8] X. Wang and H. Zeng, "History-based dynamic test case prioritization for requirement properties in regression testing," in *Continuous Software Evolution and Delivery (CSED), IEEE/ACM International Workshop on*, pp. 41–47, IEEE, 2016.

[9] M. A. Hasan, M. A. Rahman, and M. S. Siddik, "Test case prioritization based on dissimilarity clustering using historical data analysis," in *International Conference on Information, Communication and Computing Technology*, pp. 269–281, Springer, 2017.

[10] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th international conference on software engineering*, pp. 119–129, ACM, 2002.

[11] T. B. Noor and H. Hemmati, "A similarity-based approach for test case prioritization using historical failure data," in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pp. 58–68, IEEE, 2015.

[12] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 437–440, ACM, 2014.

[13] D. Marijan, "Multi-perspective regression test prioritization for time-constrained environments," in *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*, pp. 157–162, IEEE, 2015.

[14] H. Do and G. Rothermel, "A controlled experiment assessing test case prioritization techniques via mutation faults," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pp. 411–420, IEEE, 2005.

[15] S. Siddik, A. U. Gias, and S. M. Khaled, "Optimizing software design migration from structured programming to object oriented paradigm," in *16th International Conference on Computer and Information Technology (ICCIT)*, pp. 1–6, IEEE, 2014.

[16] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.